Минобрнауки России

ФГБОУ ВПО «НИУ МЭИ» АВТИ

Кафедра математического и компьютерного моделирования

**Лабораторная работа №2**

**Математическое обеспечение ЭВМ**

**«Построение лексического блока»**

**Работу выполнил:**

Солонин Егор А-14-19

Вариант 14

**Работу принял:**

Князев А. В.

Москва 2021

**Задание:**

Разработать программу, реализующую лексический блок для заданного языка.

Максимальная длина идентификатора - не менее 10 литер.

Лексический блок реализуется как подпрограмма, вызываемая синтаксическим анализатором.

Лексический блок для каждой лексемы должен указывать ее класс и значение.

Лексический блок должен строить таблицу(ы) идентификаторов и констант.

Возможны следующие виды лексем:

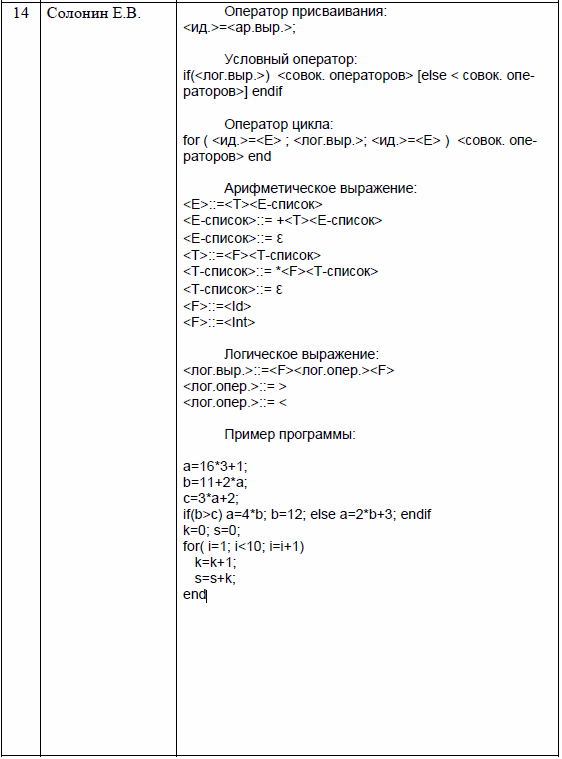
- идентификаторы;

- ключевые слова;

- целые числа;

- разделители.

Программа разрабатывается как приложение с графическим интерфейсом на языке C# в среде Visual Studio. Не должны использоваться коллекции. Не должны использоваться регулярные выражения и другие средства разбора строк.

*Индивидуальный вариант (№14):*

**Описание работы программы:**

В поле Code вносится фрагмент кода программы. При нажатии на кнопку Parse происходит построение двух таблиц – таблицы лексем и таблицы идентификаторов.

**Грамматика для лексем:**

<идент>::= буква|<идент> буква|<идент> цифра

<целое>::= цифра|<целое> цифра

<разделитель>::= +|\*|=|(|)|,|>|<|;

<ключ>::= if|else|endif|for|end

<не лекс.>::= <пробел> | \n | \r

<цифра>::= 0|1|2|3|4|5|6|7|8|9

<буква>::= a|…|z|A|…|Z

**Граф лексического блока:**

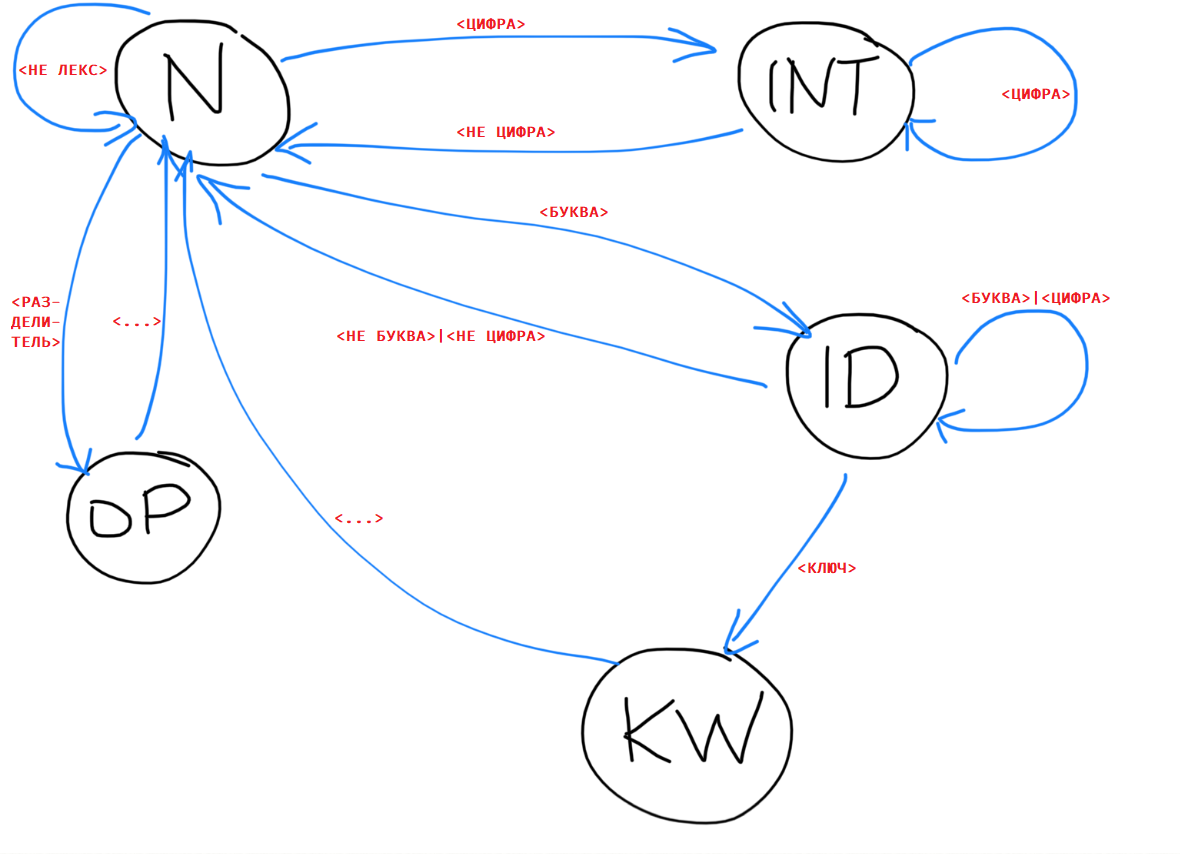
N – начальное состояние

INT – целое число

ID –идентификатор

KW – ключевое слово

OP – разделитель

****

**Алгоритмы операций на псевдокоде:**

class LexBlock – класс лексического блока, содержит:

* enum LexType – перечисление всех лексем
* char[] Terminal – все терминальные символы
* LinkedListNode table – таблица идентификаторов
* int index – свойство, указывает на текущий индекс в строке
* GetLexem(string data)– функция, позволяющая получить значение лексемы
* LexType GetLexemType(string lex) – функция, позволяющая получить тип лексемы

class Form1 : Form - содержит:

* void FillTable\_Lexems() – процедура, заполняющая таблицу лексем
* void FillTable\_Ids() – процедура, заполняющая таблицу идентификаторов

GetLexem(string data):

1. Если встречаем не лексему – переходим к следующему символу
2. Если дошли до конца строки – возвращаем \0
3. Идем по строке начиная с index
   1. Если встретили терминальный символ
      1. Если i == index – возвращаем текущий символ из строки
      2. Иначе возвращаем подстроку – все что идет после index до терминального символа
   2. Если дошли до конца строки, то возвращаем подстроку все что идет после index до конца строки
4. Возвращаем \0

LexType GetLexemType(string lex):

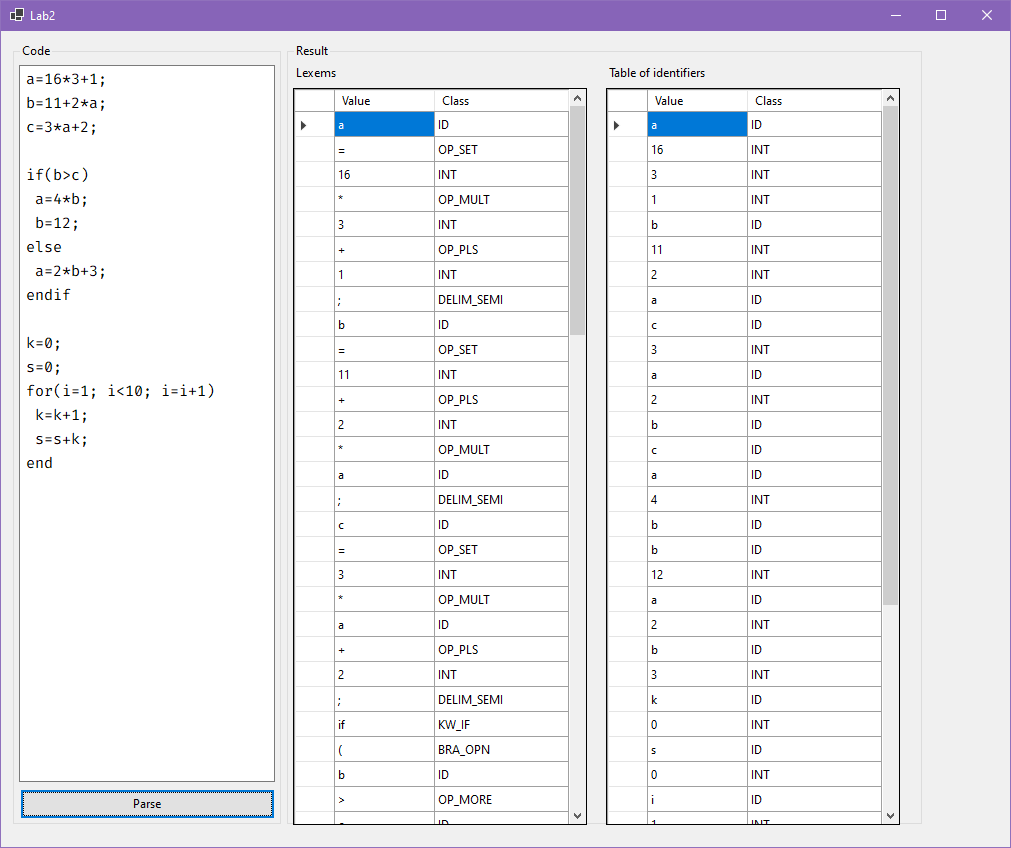
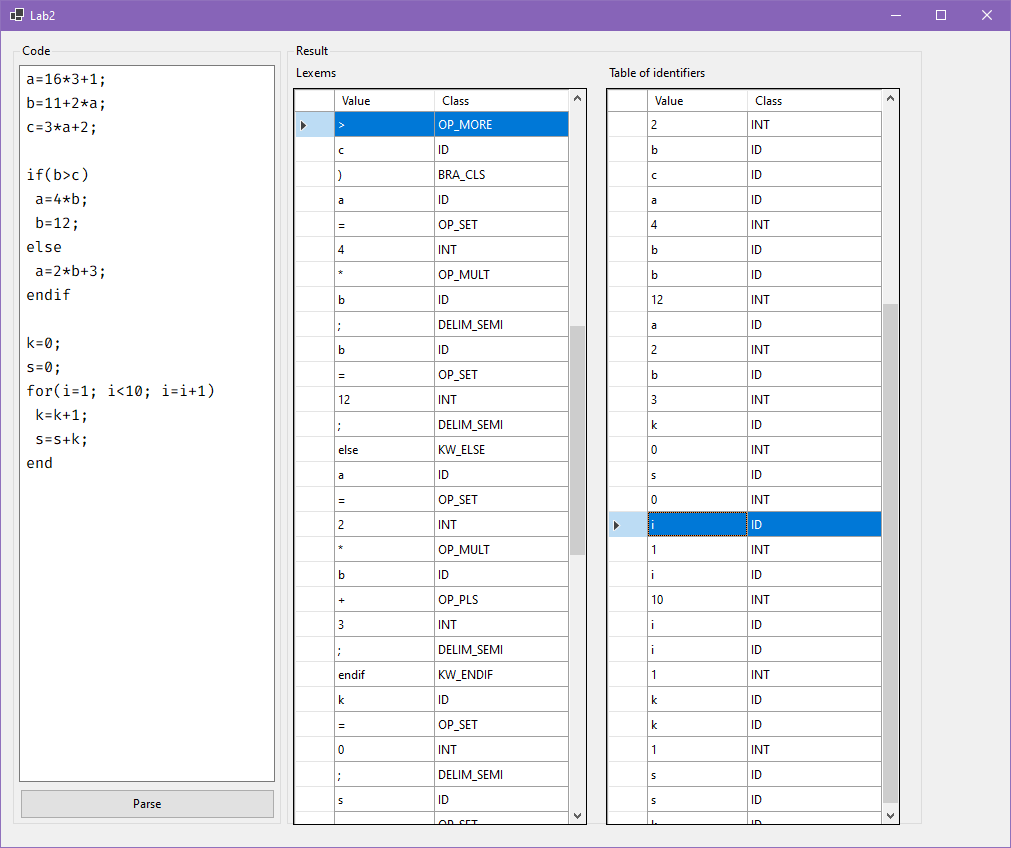
Возвращаем соответствующий тип лексемы. Отдельно обрабатываем случаи для чисел, не валидных символов и идентификаторов

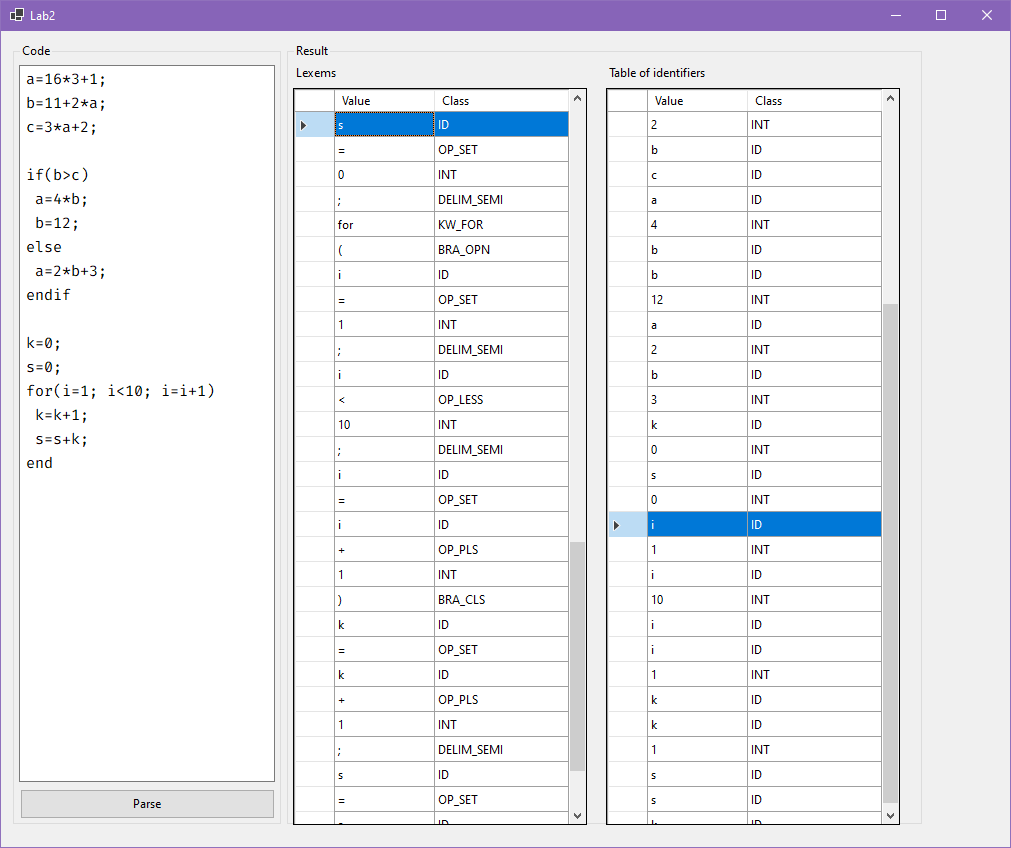
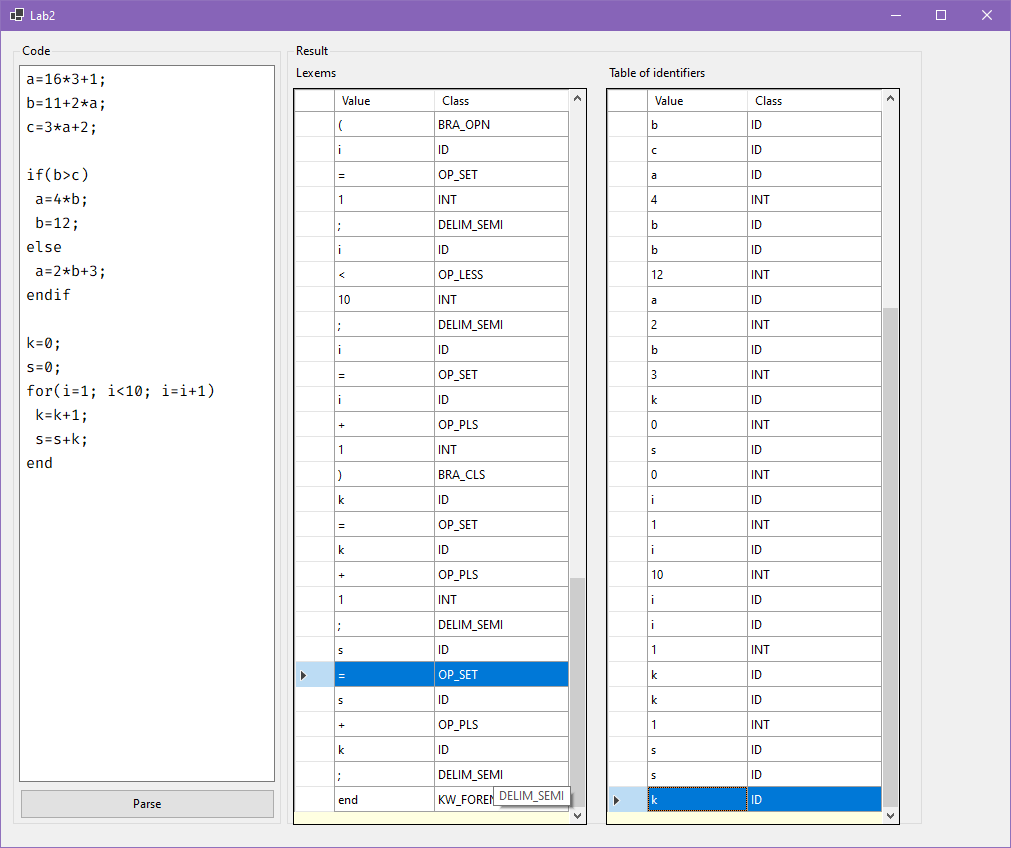
void FillTable\_Lexems():

1. Получаем очередную лексему Lexem пока не дошли до конца строки.
2. Получаем ее тип(атрибут)
3. Создаем экземпляр идентификатора и добавляем его в таблицу идентификаторов.
4. Добавляем его в таблицу лексем в форму

void FillTable\_Ids():

Идем по таблице идентификаторов и добавляем записи с типом <целое> и <идент> в таблицу в форму

**Тесты работы программы:**



**Листинг программы:**

public class Identifier {

public string Name { get; set; }

public string Attr { get; set; }

}

public class LinkedListNode {

private LinkedListNode \_head;

private LinkedListNode \_tail;

public LinkedListNode() {}

public LinkedListNode(Identifier value) {

Value = value;

}

public Identifier Value { get; internal set; }

public LinkedListNode Next { get; internal set; }

public int Count {

get;

private set;

}

public LinkedListNode Head { get { return \_head; } }

public void Add(Identifier item) {

LinkedListNode node = new LinkedListNode(item);

if(\_head == null) {

\_head = node;

\_tail = node;

} else {

\_tail.Next = node;

\_tail = node;

}

Count++;

}

public bool Contains(string name) {

LinkedListNode node = \_head;

while(node != null) {

if(node.Value.Name == name)

return true;

node = node.Next;

}

return false;

}

public bool Remove(string name) {

LinkedListNode prev = null;

LinkedListNode curr = \_head;

while(curr != null) {

if(curr.Value.Name == name) {

if(prev != null) {

prev.Next = curr.Next;

if (curr.Next == null)

\_tail = prev;

} else {

\_head = \_head.Next;

if(\_head == null)

\_tail = null;

}

Count--;

return true;

}

prev = curr;

curr = curr.Next;

}

return false;

}

public void Print()

{

LinkedListNode node = \_head;

while(node != null) {

Console.WriteLine($"{node.Value.Name}, {node.Value.Attr}");

node = node.Next;

}

}

}

static class LexBlock

{

public enum LexType

{

INVALID,

ID,

INT,

OP\_SET,

OP\_MULT,

OP\_PLS,

OP\_LESS,

OP\_MORE,

DELIM\_COM,

DELIM\_SEMI,

BRA\_OPN,

BRA\_CLS,

KW\_IF,

KW\_ELSE,

KW\_ENDIF,

KW\_FOR,

KW\_FOREND,

END,

}

public static string Input { get { return Input; } set { Input = value; index = 0; } }

public static int index { get; set; }

public static char[] Terminal = { '+','\*','(',')',',',';','=','<','>',' ','\n','\r' };

public static LinkedListNode table = new LinkedListNode();

public static string GetLexem(string data)

{

for(; (index < data.Length)

&& (data[index] == ' '

|| data[index] == '\0'

|| data[index] == '\n'

|| data[index] == '\r'); index++);

if (index == data.Length)

return "\0";

for (int i = index; i < data.Length; i++)

{

if (Array.Exists(Terminal, item => item == data[i]))

{

if (index == i)

{

index = i + 1;

return data[i].ToString();

}

string temp = data.Substring(index, i - index);

index = i;

return temp;

}

if (i == data.Length - 1)

{

string temp = data.Substring(index, i - index + 1);

index = i + 1;

return temp;

}

}

return "\0";

}

public static LexType GetLexemType(string lex)

{

switch (lex)

{

case "if": return LexType.KW\_IF;

case "else": return LexType.KW\_ELSE;

case "endif": return LexType.KW\_ENDIF;

case "for": return LexType.KW\_FOR;

case "end": return LexType.KW\_FOREND;

case "(": return LexType.BRA\_OPN;

case ")": return LexType.BRA\_CLS;

case "<": return LexType.OP\_LESS;

case ">": return LexType.OP\_MORE;

case "=": return LexType.OP\_SET;

case "+": return LexType.OP\_PLS;

case "\*": return LexType.OP\_MULT;

case ";": return LexType.DELIM\_SEMI;

case ",": return LexType.DELIM\_COM;

case "\0": return LexType.END;

default: break;

}

int num = 0;

if (int.TryParse(lex, out num))

return LexType.INT;

foreach (int l in lex)

if ((l < 'a' || l > 'z') && (l < 'A' || l > 'Z'))

return LexType.INVALID;

return LexType.ID;

}

}

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

LexBlock.table = new LinkedListNode();

LexBlock.index = 0;

FillTable\_Lexems();

FillTable\_Ids();

}

private void FillTable\_Lexems()

{

string Lexem = "";

LexemsTable.Rows.Clear();

while ((Lexem = LexBlock.GetLexem(inputData.Text)) != "\0")

{

string Attr = LexBlock.GetLexemType(Lexem).ToString();

Identifier id = new Identifier

{

Name = Lexem,

Attr = Attr

};

LexBlock.table.Add(id);

LexemsTable.Rows.Add(Lexem, Attr);

}

}

private void FillTable\_Ids()

{

TableOfIds.Rows.Clear();

LinkedListNode node = LexBlock.table.Head;

while (node != null)

{

if (node.Value.Attr == "ID" || node.Value.Attr == "INT")

TableOfIds.Rows.Add(node.Value.Name, node.Value.Attr);

node = node.Next;

}

}

}